CS422/522 Lab 3: Process Management & Trap Handling

due 2015-10-20

- Introduction
  - Getting Started
  - Hand-In Procedure
  - Inline Assembly
  - Running User Processes
- Part 1: Thread & Process Management
  - User Process Implementation in mCertiKOS
  - The PKCtxIntro Layer
  - The PKCtxNew Layer
  - The PTCBIntro Layer
  - The PTCBInit Layer
  - The PTQueueIntro Layer
  - The PTQueueInit Layer
  - The PCurID Layer
  - The PThread Layer
  - The PProc Layer
- Part 2: Trap Handling
  - Handling Interrupts and Exceptions
  - Basics of Protected Control Transfer
  - Types of Exceptions and Interrupts
  - An Example
  - Nested Exceptions and Interrupts
  - Setting Up the IDT
  - Handling Page Faults
  - System calls
  - The TSyscallArg Layer
  - The TSyscall Layer

- The TDispatch Layer
- The TTrapHandler Layer

CS422/522 Lab 3: Process Management & Trap Handling due 2015-10-20

Introduction

This lab is split into two parts. In the first part of this lab, you will implement the basic kernel facilities required to get multiple protected user-mode processes running. You will enhance the mCertiKOS kernel to set up the data structures to keep track of user processes, create multiple user processes, load a program image into a user process, and start running a user process. In the second part, you will make the mCertiKOS kernel capable of handling system calls/interrupts/exceptions made or triggered by user processes.

Getting started

In this and future labs you will progressively build up your kernel. We will also provide you with some additional source. To fetch that source, use Git to commit changes you've made since handing in lab 2 (if any), fetch the latest version of the course repository. Then create a local branch called `lab3` based on our lab3 branch, `origin/lab3`:

```
$ cd ~/cpsc422/mcertikos
$ /c/cs422/apps/syncrepo.sh
remote: Counting objects: 130, done.
remote: Compressing objects: 100% (70/70), done.
remote: Total 110 (delta 39), reused 110 (delta 39)
```

```
Receiving objects: 100% (110/110), 37.54 KiB | 0 bytes/s, done.
Resolving deltas: 100% (39/39), completed with 17 local objects.
From /c/cs422/repo/mcertikos
 * [new branch]      lab3       -> lab3
****
Remote repository synchronized successfully.
****
$ git pull
remote: Counting objects: 130, done.
remote: Compressing objects: 100% (70/70), done.
remote: Total 110 (delta 39), reused 110 (delta 39)
Receiving objects: 100% (110/110), 37.54 KiB | 0 bytes/s, done.
Resolving deltas: 100% (39/39), completed with 17 local objects.
From /c/cs422/SUBMIT/lab/xw247
 * [new branch]      lab3       -> origin/lab3
Already up-to-date.
$ git checkout -b lab3 origin/lab3
Branch lab3 set up to track remote branch refs/remotes/origin/lab3.
Switched to a new branch "lab3"
$
```

You will now need to merge the changes you made in your `lab2` branch into the `lab3` branch by `$ git merge lab2`.

In some cases, Git may not be able to figure out how to merge your changes with the new lab assignment (e.g. if you modified some of the code in Lab 2). In that case, the git merge command will tell you which files are *conflicted*, and you should first resolve the conflict (by editing the relevant files) and then commit the resulting files with `git commit -a`.

If you have trouble merging the branches, another simple solution is to copy the files you have changed in the assignment 2 into appropriate directories of assignment 3, overwriting the existing files.

It is possible that you may fail the tests for this assignment because of a bugin your code for assignment 2. We will release sample code for the assignment 2 by the end of this week (three days after its official due date). The code will be provided in a directory called `samples` under the root project directory. Please feel free to use the sample code instead of your own code. When the sample code is released, commit your local changes, and then retrive the sample code by:

```
1.  $ cd ~/cpsc422/mcertikos
2.  $ /c/cs422/apps/syncrepo.sh
3.  ****
4.  Remote repository synchronized successfully.
5.  ****
6.  $ git pull
7.  $ git push
```

* Once you've synced your remote repository, you have to run `git push` again to push your local changes back to the remote repository, since the remote one is forced to be synchronized with the updated version.

Hand-In Procedure

Include the following information in the file called `README` in the `mcertikos` directory: who you have worked with; whether you coded this assignment together, and if not, who worked on which part; and brief description of what you have implemented; and anything else you would like us to know. When you are ready to hand in your lab code, commit your changes, and then run `git push` in the `mcertikos`.

```
1.  $ git commit -am "finished lab3"
2.  [lab3 a823de9] finished lab3
3.   4 files changed, 87 insertions(+), 10 deletions(-)
4.  $ git push
```

Inline Assembly

In this lab you may find GCC's inline assembly language feature useful, although it is also possible to complete the lab without using it. At the very least, you will need to understand the fragments of the inline assembly language (`asm` statements) in the source code we gave you. You can find several sources of information on GCC inline assembly language on the class reference materials page.

Running User Processes

We have implemented a new monitor task `startuser`. Instead of the dummy process in the last assignment, once run, it starts the idle process implemented in `user/idle/idle.c`, which in turn spawns three user processes defined in `user/pingpong/ping.c`, `user/pingpong/pong.c`, and `user/pingpong/ding.c`, at the application level (ring 3), with full memory isolation and protection.

```
$> help
help - Display this list of commands
kerninfo - Display information about the kernel
startuser - Start the user idle process
$> startuser
[D] kern/lib/monitor.c:45: process idle 1 is created.
Start user-space ...
idle
ping in process 4.
pong in process 5.
ding in process 6.
ping started.
ping: the value at address e0000000: 0
ping: writing the value 100 to the address e0000000
pong started.
pong: the value at address e0000000: 0
pong: writing the value 200 to the address e0000000
```

```
ding started.
ding: the value at address e0000000: 0
ding: writing the value 300 to the address e0000000
ping: the new value at address e0000000: 100
pong: the new value at address e0000000: 200
ding: the new value at address e0000000: 300
```

We have implemented some simple procesures in those three functions to show that the memory for different processes are isolated (with page-based virtual memory), and each process owns the entire 4G of the memory. As before, the programs will not run until you have implemented all the code for the process management and trap handling modules, and the implementations of the functions from the previous assignments (or the sample code) are correct. After you can successfully run and understand the user processes we provided, you can replace it with more sophisticated user process implementations.

In the main functions of the three processes, you may notice that I explicitly call the function `yield()` to yield to another process. Since we do not have preemtion implemented in mCertiKOS, unless you explicitly yield to other processes, the current process will not release the CPU to other processes. This is obviously not an ideal situation in terms of protecting the system from bugs or malicious code in user-mode environments, because any user-mode environment can bring the whole system to a halt simply by getting into an infinite loop and never giving back the CPU. In the next assignment, we will learn how to utilize the timer hardware interrupt to allow the kernel to preempt a running process, forcefully retaking control of the CPU from it.

Part 1: Thread & Process Management

User Process Implmentation in mCertiKOS

In mCertiKOS, every process created in the application level has a corresponding service thread in the kernel. Any system call requests of a particular process will be handled by a corresponding kernel service thread. When a process yields, via the sys_yield system call, it first traps into the corresponding kernel thread; we switch the page

structure to page structure #0 (so that the kernel can access arbitrary memory via the identity page map), save the current process states (user context/trap frame), then switch to the next ready kernel service thread based on the scheduler. Then the resumed kernel thread restores the process states, switches the page structure to the corresponding process's page structure, and then jumps back to its user process.

The PKCtxIntro Layer

In this layer, we introduce the notion of kernel thread context. When you switch from one kernel thread to another, you need to save the current thread's states (the 6 register values defined in the kctx structure) and restore the new thread's states. Read `kern/thread/PKCtxIntro/PKCtxIntro.c` carefully to make sure you understand every concept.

> ### Exercise 1
>
> In the file `kern/thread/PKCtxIntro/cswitch.S`, implement the assembly function `cswitch`. This is the first exercise in this course that asks you to implement something in assembly. If you have not had chance to learn the assembly language in assignment 1, this is the perfect time to get back and review the section on Get Started with x86 Assembly in Assignment 1.
>
> In the implementation of `cswitch`, the values of all five registers except `eip`, should come from the hardware values themselves, e.g., `%edi`, `%esi`, etc. On the other hand, when you save the old kernel thread context, the `eip` value you need to save should be the return address pushed onto the current thread's stack. By the C calling convention, when function call occurs, return address is first pushed onto the stack before the arguments. Therefore, the first argument of the funciton is `4(%esp)` instead of `0(%esp)`, since the latter value represents the return address, i.e., the `eip` you need to read from or write to.

The PKCtxNew Layer

child check & set_esp different

In this layer, you are going to implement a function that creates new kernel context for a child process. Please make sure you read all the comments carefully.

> ## Exercise 2
>
> In the file `kern/thread/PKCtxNew/PKCtxNew.c`, you must implement all the functions listed below:
>
> - `kctx_new`

Testing The Kernel

We will be grading your code with a bunch of test cases, part of which are given in `test.c` in each layer sub directory. You can run `make TEST=1` to test your solutions. You can use `Ctrl-a x` to exit from the qemu.

* If you have already run `make` before, you have to first run `make clean` before you run `make TEST=1`.

```
1.  Testing the PKCtxNew layer...
2.  test 1 passed.
3.  All tests passed.
4.
5.  Testing the PTCBInit layer...
6.  test 1 passed.
7.  All tests passed.
8.
9.  Testing the PTQueueInit layer...
```

```
test 1 passed.
test 2 passed.
All tests passed.

Testing the PThread layer...
test 1 passed.
All tests passed.

Test complete. Please Use Ctrl-a x to exit qemu.
```

Write Your Own Test Cases! (optional)

Come up with your own interesting test cases to seriously challenge your classmates! In addition to the provided simple tests, selected (correct, fully documented, and interesting) test cases will be used in the actual grading of the lab assignment!

In `test.c` in each layer directory, you will find a function defined with the name `LayerName_test_own`. Fill the function body with all of your nice test cases combined. The test function should return 0 for passing the test and a non-zero code for failing the test. Be extra careful to make sure that if you overwrite some of the kernel data, they are set back to the original value. Otherwise, it may make the future test scripts fail, even if you have implemented all the functions correctly.

* Your test function itself will not be graded, so don't be afraid of submitting a wrong script.

The PTCBIntro Layer

In this layer, we introduce the thread control blocks (TCB). Since the code in this layer is fairly simple, we have already implemented it for you. Please make sure you understand all the code we provide in `kern/thread/PTCBIntro/PTCBIntro.c`.

The PTCBInit Layer

In this layer, you are going to implement a function that initializes all TCBs. Please make sure you read all the comments carefully.

> ### Exercise 3
>
> In the file `kern/thread/PTCBInit/PTCBInit.c`, you must implement all the functions listed below:
>
> - `tcb_init`

The PTQueueIntro Layer

In this layer, we introduce the thread queues. Please make sure you understand all the code we provide in `kern/thread/PTQueueIntro/PTQueueIntro.c`.

The PTQueueInit Layer

In this layer, you are going to implement a function that initializes all the thread queues, plus the functions that manipulate the queues. Please make sure you read all the comments carefully.

> ### Exercise 4
>
> In the file `kern/thread/PTQueueInit/PTQueueInit.c`, you must implement all the functions listed below:
>
> - `tqueue_init`
> - `tqueue_enqueue`
> - `tqueue_dequeue`

- `tqueue_remove`

The PCurID Layer

In this layer, we introduce the current thread id that records the current running thread id. The code is provided in `kern/thread/PCurID/PCurID.c`.

The PThread Layer

In this layer, you are going to implement a function to spawn a new thread, or to yield to another thread. Please make sure you read all the comments carefully.

> ## Exercise 5
>
> In the file `kern/thread/PThread/PThread.c`, you must implement all the functions listed below:
>
> - `thread_spawn`
> - `thread_yield`

The PProc Layer

In this layer, we introduce the functions to create a user level process. Please make sure you understand all the code we provide in `kern/proc/PProc/PProc.c`.

Part 2: Trap Handling

Handling Interrupts and Exceptions

At this point, the first `int $0x30` system call instruction in user space is a dead end: once the processor gets into user mode, there is no way to get back out. You will now need to implement basic exception and system call handling, so that it is possible for the kernel to recover control of the processor from user-mode code. The first thing you should do is thoroughly familiarize yourself with the x86 interrupt and exception mechanism.

> **Exercise 6**
>
> Read Chapter 9, Exceptions and Interrupts in the 80386 Programmer's Manual (or Chapter 5 of the IA-32 Developer's Manual), if you haven't already.

In this lab we generally follow Intel's terminology for interrupts, exceptions, and the like. However, terms such as exception, trap, interrupt, fault and abort have no standard meaning across architectures or operating systems, and are often used without regard to the subtle distinctions between them on a particular architecture such as the x86. When you see these terms outside of this lab, the meanings might be slightly different.

Basics of Protected Control Transfer

Exceptions and interrupts are both "protected control transfers," which cause the processor to switch from user to kernel mode (CPL=0) without giving the user-mode code any opportunity to interfere with the functioning of the kernel or other environments. In Intel's terminology, an *interrupt* is a protected control transfer that is caused by an asynchronous event usually external to the processor, such as notification of external device I/O activity. An *exception*, in contrast, is a protected control transfer caused synchronously by the currently running code, for example due to a divide by zero or an invalid memory access.

In order to ensure that these protected control transfers are actually *protected*, the processor's interrupt/exception

mechanism is designed so that the code currently running when the interrupt or exception occurs *does not get to choose arbitrarily where the kernel is entered or how*. Instead, the processor ensures that the kernel can be entered only under carefully controlled conditions. On the x86, two mechanisms work together to provide this protection:

1. **The Interrupt Descriptor Table.** The processor ensures that interrupts and exceptions can only cause the kernel to be entered at a few specific, well-defined entry-points *determined by the kernel itself*, and not by the code running when the interrupt or exception is taken.

   The x86 allows up to 256 different interrupt or exception entry points into the kernel, each with a different *interrupt vector*. A vector is a number between 0 and 255. An interrupt's vector is determined by the source of the interrupt: different devices, error conditions, and application requests to the kernel generate interrupts with different vectors. The CPU uses the vector as an index into the processor's *interrupt descriptor table* (IDT), which the kernel sets up in kernel-private memory, much like the GDT. From the appropriate entry in this table the processor loads:

   1. the value to load into the instruction pointer (`EIP`) register, pointing to the kernel code designated to handle that type of exception.
   2. the value to load into the code segment (`CS`) register, which includes in bits 0-1 the privilege level at which the exception handler is to run. (In mCertiKOS, all exceptions are handled in kernel mode, privilege level 0.)
2. **The Task State Segment.** The processor needs a place to save the *old* processor state before the interrupt or exception occurred, such as the original values of `EIP` and `CS` before the processor invoked the exception handler, so that the exception handler can later restore that old state and resume the interrupted code from where it left off. But this save area for the old processor state must in turn be protected from unprivileged user-mode code; otherwise buggy or malicious user code could compromise the kernel.

   For this reason, when an x86 processor takes an interrupt or trap that causes a privilege level change from user to kernel mode, it also switches to a stack in the kernel's memory. A structure called the *task state segment* (TSS) specifies the segment selector and address where this stack lives. The processor pushes

(on this new stack) `SS`, `ESP`, `EFLAGS`, `CS`, `EIP`, and an optional error code. Then it loads the `CS` and `EIP` from the interrupt descriptor, and sets the `ESP` and `SS` to refer to the new stack.

Although the TSS is large and can potentially serve a variety of purposes, mCertiKOS only uses it to define the kernel stack that the processor should switch to when it transfers from user to kernel mode. Since "kernel mode" in mCertiKOS is privilege level 0 on the x86, the processor uses the `ESP0` and `SS0` fields of the TSS to define the kernel stack when entering kernel mode. mCertiKOS doesn't use any other TSS fields.

If interested, read the related code in `kern/lib/seg.c`

Types of Exceptions and Interrupts

All of the synchronous exceptions that the x86 processor can generate internally use interrupt vectors between 0 and 31, and therefore map to IDT entries 0-31. For example, a page fault always causes an exception through vector 14. Interrupt vectors greater than 31 are only used by *software interrupts*, which can be generated by the `int` instruction, or asynchronous *hardware interrupts*, caused by external devices when they need attention. mCertiKOS (fairly arbitrarily) uses software interrupt vector 48 (0x30) as its system call interrupt vector.

An Example

Let's put these pieces together and trace through an example. Let's say the processor is executing code in a user environment and encounters a divide instruction that attempts to divide by zero.

1. The processor switches to the stack defined by the `SS0` and `ESP0` fields of the TSS, which in mCertiKOS will hold the values `GD_KD` and `KSTACKTOP`, respectively.
2. The processor pushes the exception parameters on the kernel stack, starting at address `KSTACKTOP`:

```
1.                  +--------------------+ KSTACKTOP
2.                  | 0x00000 | old SS   |     " - 4
3.                  |      old ESP       |     " - 8
4.                  |     old EFLAGS     |     " - 12
```

```
                    | 0x00000 | old CS    |      " - 16
                    |      old EIP        |      " - 20 <---- ESP
                    +---------------------+
```

1. Because we're handling a divide error, which is interrupt vector 0 on the x86, the processor reads IDT entry 0 and sets `CS:EIP` to point to the handler function described by the entry.
2. The handler function takes control and handles the exception, for example by terminating the user environment.

For certain types of x86 exceptions, in addition to the "standard" five words above, the processor pushes onto the stack another word containing an *error code*. The page fault exception, number 14, is an important example. See the 80386 manual to determine for which exception numbers the processor pushes an error code, and what the error code means in that case. When the processor pushes an error code, the stack would look as follows at the beginning of the exception handler when coming in from user mode:

```
                    +---------------------+ KSTACKTOP
                    | 0x00000 | old SS    |      " - 4
                    |      old ESP        |      " - 8
                    |     old EFLAGS      |      " - 12
                    | 0x00000 | old CS    |      " - 16
                    |      old EIP        |      " - 20
                    |     error code      |      " - 24 <---- ESP
                    +---------------------+
```

Nested Exceptions and Interrupts

The processor can take exceptions and interrupts both from kernel and user mode. It is only when entering the kernel from user mode, however, that the x86 processor automatically switches stacks before pushing its old register state onto the stack and invoking the appropriate exception handler through the IDT. If the processor is *already* in kernel mode when the interrupt or exception occurs (the low 2 bits of the `CS` register are already zero),

then the CPU just pushes more values on the same kernel stack. In this way, the kernel can gracefully handle *nested exceptions* caused by code within the kernel itself. This capability is an important tool in implementing protection.

On the other hand, in the version of mCertiKOS provided in this lab, we always turn off the interrupts when we are in the kernel mode, to make our life simpler.

Setting Up the IDT

You should now have the basic information you need in order to understand the code for the IDT set up and exception handling in mCertiKOS. Related code is in `lib/trap.h`, `dev/intr.h`, `dev/intr.c`, `dev/idt.S`.

Note: Some of the exceptions in the range 0-31 are defined by Intel to be reserved. Since they will never be generated by the processor, it doesn't really matter how you handle them. So we can do whatever we think is cleanest.

The overall flow of control that you should achieve is depicted below:

```
1.          IDT                    dev/idt.S          trap/TTrapHandler/TTrapHandle
    r.c
2.
3.    +-----------------+
4.    |   &handler1     |---------> handler1:          trap (struct tf_t *tf)
5.    |                 |             // do stuff      {
6.    |                 |             call trap          // handle the exception/int
    errupt
7.    |                 |             // ...           }
8.    +-----------------+
9.    |   &handler2     |--------> handler2:
10.   |                 |            // do stuff
11.   |                 |            call trap
12.   |                 |            // ...
```

```
+-----------------+
        .
        .
        .
+-----------------+
|   &handlerX     |--------> handlerX:
|                 |              // do stuff
|                 |              call trap
|                 |              // ...
+-----------------+
```

Each exception or interrupt should have its own handler in `dev/idt.S` and `intr_init_idt()` in `dev/intr.c` should initialize the IDT with the addresses of these handlers. Each of the handlers should build a `struct tf_t` (see `lib/trap.h`) on the stack and call `trap()` (in `trap/TTrapHandler/TTrapHandler.c`) with a pointer to the trap frame. `trap()` then handles the exception/interrupt or dispatches to a specific handler function.

Handling Page Faults

The page fault exception, interrupt vector 14 (`T_PGFLT`), is a particularly important one. When the processor takes a page fault, it stores the linear (i.e., virtual) address that caused the fault in a special processor control register, `CR2`. In `trap/TTrapHandler/TTrapHandler.c` we have provided the implementation of `pgflt_handler()`, to handle page fault exceptions.

System calls

User processes ask the kernel to do things for them by invoking system calls. When the user process invokes a system call, the processor enters kernel mode, the processor and the kernel cooperate to save the user process's state, the kernel executes appropriate code in order to carry out the system call, and then resumes the user process. The exact details of how the user process gets the kernel's attention and how it specifies which call it wants to execute vary from system to system.

In the mCertiKOS kernel, we will use the `int` instruction, which causes a processor interrupt. In particular, we will use `int $0x30` as the system call interrupt. We have defined the constant `T_SYSCALL` to 48 (0x30) for you. You will have to set up the interrupt descriptor to allow user processes to cause that interrupt. Note that interrupt 0x30 cannot be generated by hardware, so there is no ambiguity caused by allowing user code to generate it.

The application will pass the system call number and the system call arguments in registers. This way, the kernel won't need to grub around in the user environment's stack or instruction stream. The system call number will go in `%eax`, and the arguments (up to five of them) will go in `%edx`, `%ecx`, `%ebx`, `%edi`, and `%esi`, respectively. A system call always returns with an error number via register EAX. All valid error numbers are listed in __error_nr defined in `kern/lib/syscall.h`. E_SUCC indicates success (no errors). A system call can return at most 5 32-bit values via registers EBX, ECX, EDX, ESI and EDI. When the trap happens, we first save the corresponding trap frame (the register values of the user process) into memory (`uctx_pool`), and we restores the register values based on the saved ones.

The implementation of the system call library in the user level, following the calling conventions above, can be found in `user/include/syscall.h`. In this part of the assignment, you will implement various kernel functions to handle the user level requests inside the kernel.

The TSyscallArg Layer

In this layer, you are going to implement the functions that retrives the arguments from the user context, and ones that sets the error number and return values back to the user context, based on the calling convention described above.

> ## Exercise 7
>
> In the file `kern/trap/TSyscallArg/TSyscallArg.c`, you must implement all the functions listed below. Note that `syscall_get_arg1` should return the system call number (`eax`), not the actual first argument of the function (`ebx`).
>
> - `syscall_get_arg1`

- `syscall_get_arg2`
- `syscall_get_arg3`
- `syscall_get_arg4`
- `syscall_get_arg5`
- `syscall_get_arg6`
- `syscall_set_errno`
- `syscall_set_retval1`
- `syscall_set_retval2`
- `syscall_set_retval3`
- `syscall_set_retval4`
- `syscall_set_retval5`

The TSyscall Layer

Exercise 8

In the file `kern/trap/TSyscall/TSyscall.c`, you must correctly implement all the functions listed below:

- `sys_spawn`
- `sys_yield`

The TDispatch Layer

This layer implements the function that dispatches the system call requests to appropriate handlers we have implemented in the previous layer, based on the system call number passed in the user context. Make sure you

fully understand the code in `kern/trap/TDispatch/TDispatch.c`.

The TTrapHandler Layer

> ## Exercise 9
>
> In the file `kern/trap/TTrapHandler/TTrapHandler.c`, carefully review the implementation of the function `trap`, then implement all the functions listed below:
>
> - `exception_handler`
> - `interrupt_handler`

This completes the lab. Make sure you pass all of the `make TEST=1` tests and don't forget editing your `README` file. Some layers may not have test cases implemented. So make sure the three user programs run smoothly after everything is implemented. Feel free to experiment with different implementations of user processes. Commit your changes and type `git push` in the `mcertikos` directory to submit your code.